Synchronizing processors through memory requests in a tightly coupled multiprocessor

André Seznec, Yvon Jégou

INRIA

UNITÉ DE RECHERCHE
INRIA-RENNES

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P.105
78153 Le Chesnay Cedex
France
Tél.:(1) 39 63 55 11

Rapports de Recherche

N° 762

SYNCHRONIZING PROCESSORS THROUGH MEMORY REQUESTS IN A TIGHTLY COUPLED MULTIPROCESSOR

André SEZNEC
Yvon JEGOU

NOVEMBRE 1987

IRISA INSTITUT DE RECHERCHE EN INFORMATIQUE ET SYSTÈMES ALÉATOIRES

Campus Universitaire de Beaulieu
35042 - RENNES CÉDEX
FRANCE

Synchronisation des processeurs par les requêtes à la mémoire dans un multiprocesseur fortement couplé

Synchronizing processors through memory requests in a tightly coupled multiprocessor

André Seznec
Yvon Jégou
IRISA/INRIA, Campus de Beaulieu
35042 Rennes Cedex
FRANCE

Publication Interne n° 380 - Novembre 87 - 24 pages

Résumé :

Aujourd'hui, les supercalculateurs (et les minisupercalculateurs) sont des multiprocesseurs dont les processeurs sont des processeurs pipelines. Les performances effectives de ces calculateurs dépendent essentiellement du spectre d'algorithmes qui sont effectivement exécutés en parallèle.

Nous avons présenté précédemment le processeur DSPA [Je86] et le réseau GLOUTON (GREEDY) [Se86][Se87c] qui permet d'introduire le processeur DSPA comme processeur élémentaire d'un multiprocesseur.

Dans ce rapport, nous introduisons des mécanismes de synchronisation plus simples (et plus faciles à mettre en oeuvre) que ceux présentés dans [Se86][Se87c]. Quand des processeurs DSPA sont connectés à une mémoire partagée à travers des réseaux GLOUTON et synchronisés par nos mécanismes, un haut degré de parallélisme peut être obtenu à l'exécution sur une très large famille d'algorithmes, y compris des boucles où les dépendances entre les itérations ne peuvent être détectés à la compilation.

Abstract

To satisfy the growing need for computing power, a high degree of parallelism will be necessary in future supercomputers. Up to the late 70s, supercomputers were either multiprocessors (SIMD-MIMD) or pipelined monoprocessors. Current commercial products combine these two levels of parallelism.

Effective performance will depend on the spectrum of algorithms which is actually run in parallel. In a previous paper [Je86], we have presented the DSPA processor, a pipeline processor which is actually performant on a very large family of loops.

In this paper, we present the GREEDY network, a new interconnection network (IN) for tightly coupled multiprocessors (TCMs). Then we propose an original and cost effective hardware synchronization mechanism. When DSPA processors are connected with a shared memory through a GREEDY network and synchronized by our synchronization mechanism, a very high parallelism may be achieved at execution time on a very large spectrum of loops including loops where independency of the successive iterations cannot be checked at compile time as e.g. loop 1 :

```
DO 1 I=1,N
1 A(P(I))=A(Q(I))
```

CENTRE NATIONAL DE LA RECHERCHE SCIENTIFIQUE
(L. A. 227)
UNIVERSITÉ DE RENNES 1 I. N. S. A. DE RENNES

INSTITUT NATIONAL DE RECHERCHE
EN INFORMATIQUE ET EN AUTOMATIQUE
(LABORATOIRE DE RENNES)

Introduction

Needs for performance in various scientific applications have lead manufacturers to the design of parallel structures. The first industrial parallel supercomputers were pipeline processors (Cray1, CDC Cyber 205, ..). Now, supercomputers (Cray 2, ETA^{10}, ..) combine at least two levels of parallelism; they are multiprocessors where each processor is a powerful pipeline processor.

In many applications, effective performance on supercomputers does not depend on the peak performance but on the spectrum of algorithms which are actually efficiently run in parallel. In [Je86], we have presented a pipeline model, the Data Synchronized Pipeline Architecture (DSPA). This pipeline processor has been shown to be actually efficient on the whole set of imperative loops i.e. the loops which cannot be forgiven before the loop index has reached its upper bound —classical vector loops and also many loops where independency of the successive iterations cannot be checked at compile time (e.g. loop 1 and loop 2).

```
Loop 1                  Loop 2
  DO 1 I=1,N              DO 2 I=1,N
1 A(P(I))=A(Q(I))       2 A(P(I))=A(P(I))+X(L(I))*Y(I)
```

Loop 1 is generally considered as a completely scalar loop; in this paper, we propose an architecture of a TCM where loop 1 will be run in parallel : performance is only limited by the effective dependencies and the theoretical memory throughput.

I Address Synchronized Multiprocessor

1 Passing the WRITEs by the READs

The DSPA processor [Je86] is based on asynchronous independent functional units communicating through FIFO queues. We have shown that passing the WRITEs by the READs on memory with hardware detection of RAW hazards allows this pipeline processor to achieve good performance on a large family of loops where independency between successive iterations cannot be detected at compile time —e.g loop 1 and 2. In this paper, we consider the introduction of a DSPA-like processor as basic processor in a TCM. The only characteristic of the DSPA processor which has to be kept in mind is :
The production of addresses and the completing of the associated requests are dissociated.

For more details on the DSPA processor, see [Je86].

In a TCM, there are two alternative locations where the hardware detection of RAW hazards may be performed :

□ A) A hardware mechanism to detect RAW dependencies is implemented in each processor. A read request from processor I is sent to the memory if and only if there is no waiting write request on the same location *from processor I*. Write requests are sent to the memory when their associated data is available.

□ B) The RAW hazard detection is performed in the memory banks —the memory banks are assumed to have only one request port. Requests are sent to the memory; read addresses are checked against the addresses of all the WRITEs waiting for their associated data in the memory bank.

The first solution allows to send synchronously the write requests and their associated data on the memory; only two INs are needed : a (request+ write) IN and a read IN. Requests and data to be written onto memory are routed through the request IN from the processors to the memory banks. Read data are routed from the memory to the processors through the read IN. Unfortunately, spreading loops among the processors would be difficult :

- Loop 2 cannot be spread among the processors without using complex synchronization primitives to ensure consistency of the execution : when processor I reads an element of array A, a mechanism must guarantee that the associated WRITE of the same element is completed before an other processor reads the element.
- Chaining two successive vector loops may require the completeness of the WRITEs on the memory as in the following example when P and/or K are ignored at compile time :

```
      DO 20 I=1,N
20  A(I+K)= ..
      DO 21 I=1,M
21  ..  = A(I+P) ..
```

From now, in this paper, we consider that an hardware detection of the RAW hazards is performed in the global memory. The signification of a sequence of requests on a memory bank is completely determined by the order the requests enter the memory bank; we shall see in section III that this allow to synchronize the processors. For convenience, we will refer to a multiprocessor where a hardware detection of RAW hazards is performed in the global memory as an *Address Synchronized Multiprocessor* (ASM).

On an ASM, when a Read-Modify-Write instruction is implemented in the instruction set of the processor, loop 2 may be spread among the processors; each processor executes a vector loop and no synchronization of the processors is needed during this execution.

On an ASM, the sending of a write request and the sending of the associated datum are dissociated. Then three INs are needed :

- a request IN
- a read IN
- a write IN

2 Avoiding deadlocks on the memory in an ASM

In a TCM, at every cycle any processor may desire to send a request to any memory bank. Requests of several processors may have the same destination bank at the same cycle. A memory bank can process only one request at each cycle. Destination conflicts must be treated; only one request is accepted on each destination bank, the other requests must be rejected or delayed.

Let us consider the following example :

Processor 1	Processor 2
READ A	READ B
WRITE B	WRITE A

Let us now suppose that A is located in bank 0 and B is located in bank 1. We also suppose that the computation of the new value of B by processor 1 requires the value of A and that the computation of the new value of A by processor 2 requires the value of B.

Because in an ASM, the sending of the data to be written is dissociated with the sending of write requests, the following distribution of the requests on the memory banks may be logically possible but *must be avoided :*

WRITE A enters bank 0 before READ A *and* WRITE B enters bank 1 before READ B.

If this situation occurs, there would be a deadlock on the memory : WRITE requests are waiting for their associated data which cannot be computed.

Multistage networks with internal memorization of the rejected requests such as proposed for the RP3 project [Pf85] or the NYU [Go83] cannot be used in an ASM because the previous situation may occur.

Synchronous networks allow to overpass this difficulty under the condition that if a request from processor I is rejected, it is submitted again before another request from the same processor is submitted.

We give here some external characteristics of the arbiter for a synchronous IN in a TCM where a global hardware detection of RAW hazards is performed :

- N*(log N +1) input bits (the destinations of the requests)
- (N-1)*N/2 comparators of (log N +1) bits width (one for each pair of destinations)
- N output bits for validation. (one bit by request)
- An arbitration decision must be taken in one single memory cycle.

These characteristics limit the size of the IN about twenty or thirty inputs and outputs.

3 A hardware difficulty

In an ASM, the response time of a memory bank for a READ is not constant : it depends on whether the request has been delayed by a RAW hazard or not. A word of data read on bank i by processor k may flow out from the read IN before a word of data read on bank j even when the second request is older than the first request. When using a synchronous IN, there are no means to avoid this difficulty. When flowing from the read IN, data must be stored in an intermediate buffer and reordered.

In this buffer, a FIFO queue associated with each memory bank has to be implemented. This complex reordering mechanism has to be implemented in each processor. A similar mechanism must also be implemented in each memory bank to re-associate the data to be written with the corresponding write requests.

The cost of the implementation of an ASM would be prohibitive if synchronous INs are used —INs + arbiters + mechanisms for reordering data in memory banks and in processors. In the next section, we propose a new IN, the GREEDY network. The GREEDY network is virtually conflict free and allows a realistic implementation of an ASM.

II The GREEDY network

1.Definition

Definition (fig.1):

A N*M GREEDY network (or G-network for short) is a N*M array of FIFO queues where FIFO (i,j) is written from inlet i and read into outlet j, independently from other FIFO queues.

First we notice that a G-network can accept a word of data from each input during each cycle and a word of data may flow out from each output of the network during each cycle. When a FIFO queue is full, the G-network must delay the acceptance of a new word of data, but it is noteworthy that this congestion is local : it is the state of one FIFO queue, the other FIFO queues may continue to work independently.

If the definition of the G-network is quite simple, it does not solve the problem of controlling of the network.

2 Controlling the memory request G-network

Let K be the number of processors. In each memory bank i, there is a memory sequencer unit, MSU(i). At each cycle, MSU(i) receives from each processor j a 1-bit information b(i,j). The information "a request from processor j concerns memory bank i" is coded in bit b(i,j). The whole vector of informations is stored in a FIFO queue (width K bits, one bit by processor) in the MSU; an immediate optimization consists in storing this vector only if there is some request concerning the memory bank.

Processing requests in a memory bank B is then straightforward (fig.2) :

Step 1. Extract a K-bit vector V from the FIFO queue if it is not empty. Otherwise GOTO step 1.

Step 2. If vector V is null then GOTO step 1. Otherwise

X <-- number of the first nonnull bit

V(X)=0

Step 3. Extract a request from the FIFO queue (X,B) in the G-network and begin processing it. GOTO step 2.

The hardware required in the MSU to perform this treatment is limited to a FIFO queue and a priority encoder.

3 P-issuing : definition

Notice that a non-null validation bit b(j,i) must correspond to each request originated from processor j to bank i and vice-versa.

Definition :

A memory request destinated to memory bank i will be said P-issued when the MSU of the requested memory bank has received the associated validation bit.

Notice that the P-issuing of a request may be dissociated with the deposit of the request in the request G-network.

4 A condition for avoiding deadlock on the memory

As the order of the READs automatically corresponds to the order of the WRITEs on a FIFO queue, requests from processor j to a single memory bank i are automatically P-issued in the logical order of the programs.

Theorem 1 :

Let us suppose that :

(A) If a request R1 originated from processor j is younger than a request R2 originated from the same processor, then R1 cannot be P-issued *before* R2.

(B) In a program, a WRITE is not requested before all the READs of the operands needed for the computation of the word of data to be written have been requested.

Then no deadlocks can occur on the memory.

Proof :

First we give a label to each request. Let K be the number of processors.

If a request from processor j to bank i is P-issued at cycle t_1 and has been computed at cycle t_0, the request gets the label (Kt_1+j, t_0).

We define a complete order on the set of labels by : $(x,y) << (x',y')$ iff $x<x'$ or $(x=x'$ and $y < y')$.

It is obvious that memory bank i processes its requests in the increasing order of their labels.

We consider the whole set S of requests which have already been P-issued and have not been completed at an instant T. S contains requests which are always present in the request G-network and the write requests which have already entered the RAW detection mechanism in the memory banks and are waiting for their associated data.
If S is not empty then it exists a request R in S which label is minimum.
If the request R is a READ, it may be performed at soon as the desired bank becomes free.
If the request R is a WRITE then, condition (B) and condition (A) guarantee that the READs of the operands needed for the computation of the word of data to be written have labels smaller than the label of R : they have been completed. Then the word of data associated with the request R will be available in a finite delay.
Then no deadlocks are possible on the memory.

Q.E.D.

Notice that several memory requests from processor j destinated to distinct memory banks may be P-issued *at the same time.*

Definition :

A memory request will be called a dummy request if it is not destinated to any memory bank — synchronization requests for example.

Condition (A) allows to define the P-issuing of a dummy request as follows :
A dummy request from processor j will be said to have been P-issued as soon as a younger request from processor j may be P-issued.

In section III, we shall show that processors may be efficiently synchronized through the P-issuing of the memory requests and we present an efficient P-issuing hardware mechanism.

5 Controlling the read G-network and the write G-network

Controlling the read network and the write network is straightforward. Here we describe in detail the control of the read G-network. A similar mechanism can be used to control the write G-network.

In the processor, data are obtained from the read network by the Data Acquisition Unit (DAU) (fig.3). When a read request is deposited into the request network, the label of the addressed memory bank is stored in a FIFO queue located in the DAU. To obtain the words of data coming back from the memory in the same order as their addresses have been computed, the DAU reads its FIFO queue and uses this value as an address to select the FIFO queue in the read G-network from which the word of data will be available. The DAU waits until this FIFO queue is not empty.

It is noteworthy that the DAU is a simple device. Let us notice that the delay to obtain a data from a memory bank is not constant. The DAU complexity favorably compares with the complexity of hardware mechanism which would be necessary to reorder the data flowing from a crossbar network. In the latter case, data would have to be stored in an intermediate buffer and reordered. In this buffer, K FIFO queues would have to be simulated. Moreover when using crossbar networks as request, read and write networks, many bits of information must be transmitted through the networks with the data to re-associate data and requests : label of the origin processor for the request and the write networks, label of the origin memory bank for the read network.

6 Feasibility of the GREEDY network

FIFO queue components with cycle around 50 ns are proposed by several societies. These FIFO queues are generally 9 bits wide and very deep (256 words or even more). The design of a 16*16 G-network would require the use of 256 FIFO queues. On a supercomputer, the width of the word of data must be 64 bits. Then the read network must be 64 bit wide. The widths of the request network and the write network may be limited to 32 bits. So 4,096 FIFO queue cells are needed for the design of the INs of an ASM with 16 memory banks and 16 processors; the size of this network may be considered as a major difficulty.

The characteristics of the available FIFO queue components do not match with our application. They are very deep but not very wide. Our idea was to implement many FIFO queues on the same chip. A 4*4 G-network of 12 bits width FIFO queues and about 24 words by FIFO queue can be implemented on a single 144-pin chip with 65,000 transistors [Co86]. This chip will accept a WRITE from each input *and* a READ onto each output every 50 ns. This basic chip can be used to built wider and larger G-networks. For our application, only 192 basic chips will be needed in the design of the three INs.

7 The GREEDY network and a shared interleaved and pipelined memory

Large static memory components with cycles around 50 ns are now available and may be used in the design of an interleaved memory. But the cost of a very large memory built with such components will be prohibitive —these fast chips are expensive. It seems more attractive to use memory chips with a longer memory cycle. These chips are cheaper and have a very high density of integration. This approach has been adopted by Cray Research for the design of the Cray-2. Memory chips with 120 ns cycle have been used in the design of the Cray-2 memory; the size of this memory is 256 Megawords of 64 bits against only 8 Megawords in the Cray-XMP where memory chips with 40 ns cycle are used.

Let us consider a TCM with 16 pipeline processors which are able to initiate a memory request every 50 ns and chips of dynamic RAM with cycles around 300 ns. The memory of this multiprocessor would be divided in 96 or 128 memory banks (for easier address computation) in order to obtain a

theoretical throughput equal or higher than one word per 50 ns and processor. The cost of a 128*16 G-network would be huge. During one cycle of 50 ns, at most 16 banks receive a data. As it takes more than one cycle on a memory bank to treat a request, several memory banks may share the same output (or input) of the IN. If the read data flow out from this logical bank in the same order their READs have been requested, this group of physical memory banks may be considered as only one logical unit. We call this logical unit a logical memory bank.

In this section we present an optimized organization of a logical memory bank [Je86][Se87a] (fig.4).

Each physical memory bank i has a FIFO queue of requests R(i) and a FIFO queue DI(i) as inputs and a FIFO queue DO(i) as output. Memory requests (resp. data to be written) flow from the memory sequencing unit (MSU) of the logical memory bank in the order of their arrivals on the logical memory bank and are written in the required R(i) (resp. DI(i)). If the request is a READ then the label of the requested physical bank is stored in the FIFO queue F. Bank i loads a request from R(i), if the request is a WRITE, the address is stored in some associative memory until the associated data is valid —the clock of the associative memory may be in the range of 300 ns. If the request is a READ, the address is checked against this associative memory; if there is some WRITE on the same memory location waiting for its data, then the physical bank is blocked until this WRITE has been completed, otherwise the READ is performed and the read data is deposited in DO(i). Then the reordering unit (RU) uses the FIFO queue F as address to read the data on the FIFO queues DO(i). The data flow out from the logical memory bank in the desired order.

We have simulated by software the behavior of a memory consisting in 16 logical banks accessed through G-networks. The following hypothesis have been assumed :

- 16 processors
- 8 physical banks by logical bank
- a physical memory bank is busy during T_{busy}=6 cycles by a request
- Maximum depth of the FIFO queues of requests and data : 16
- Maximum depth of the FIFO queues in the G-network : 16
- All the requests are read requests
- The requests are randomly distributed on the 128 physical banks
- A new request is presented by each processor at each cycle if the previous request has been accepted by the request G-network.

The effective throughput is 97% of the theoretical throughput of the memory. When using the blocking structure of logical memory bank and crossbar networks, only 31% of the theoretical

throughput is obtained.

These results prove that the association of GREEDY networks with the proposed structure of a logical memory bank may be used to built a very large memory with relatively slow (and cheap) dynamic RAM components. Its effective throughput allows to consider this solution as a very promising alternative to the use of memory caches; many difficulties are then avoided : cache coherence, need of data locality..

Remark : The lack of conflicts in the distribution of consecutive elements of a vector among the memory banks does not remain the golden rule for the mapping of data in memory [La75][Bu71]. In [Se87b], we propose a new mapping of data in a 2^n -way interleaved memory. This mapping allows to use the whole theoretical memory throughput during accesses to constant-strided vectors with strides of the form $2^r t$ where t is odd and where 2^r is smaller or equal to the depth of the FIFO queues in the G-network.

In the next section, we answer to the following question :

— How can we synchronize the processors in an ASM without paying a large time penalty ?

III A proposition for synchronizing the processors in an ASM

The performance of algorithms on a TCM depends on the microtasking facilities of the architecture. Here we propose a synchronization of the processors limited to a synchronization on the P-issuing of the memory requests.

First in section III.1, we define four kinds of memory requests and we show that these families of requests allows the parallel execution of a very large family of algorithms. Then in section III.2, we propose hardware mechanisms which allows a very efficient P-issuing for the previous four types of memory requests.

1 Four types of memory requests

1.1 Standard requests

We first define standard memory requests :

Definition :

There are no synchronization on the P-issuing of standard requests, the P-issuing of standard requests has only to respect the condition (A) in theorem 1.

Let us consider the whole family of loops whose iterations are independent from each other i.e loops which can be transformed in :

```
DO 30 I=1,N
```

30 CALL TASK(I)

where only local variables are modified by TASK(I) i.e. if a variable is written by TASK(I), it is not read by TASK(J) for J≠I.

The iterations of these loops can be distributed among the processors and only standard memory requests are used for coding the loop body.

For example, iterations of vectorized sequential loops can be coded using only standard requests. But in many cases, because of data dependencies, some global synchronization is required for chaining successive loops (e.g. loops 20 and 21). On many classical multiprocessors, such synchronizations are quite CPU time consuming and often drop the efficiency of the computation on short vector loops.

1.2 Global synchronization requests

We define global synchronization requests as follows :

Definition :

A global synchronization request (GSR) from processor j is P-issued when and only when all the processors have to P-issue a global synchronization request.

GSRs allow to implement global joins without interrupting the sequencing of the processors. We shall see in section III.2 that GSRs even do not interrupt the computations of memory requests.

— GSRs may be used in order to enforce the respect of dependencies when chaining two successive loops (eg. loops 20 and 21). A dummy GSR is executed by each processor after the last memory request of its last iteration of the first loop.

— GSRs may also be useful to parallelize loops including indirect WRITEs accesses.

— GSRs may be used for the read of the same data by all the processors (for example the number of iterations or a constant operand for a vector loop). All the processing elements synchronize on the P-issuing of this read request. The read request of processor 0 is really P-issued, the other processors P-issue a dummy request, but the read data is broadcast to all the processors. This avoids K-1 read requests conflicting on the same memory location.

1.3 Master requests and slave requests

1.3.1 A family of imperative loops

Notice that standard requests and GSRs allow to process in parallel on an ASM a large family of algorithms. All the loops which are considered as "vector" loops for a DSPA processor may be processed in parallel : loop 2 is a vector loop [Je86][Se87a].

Nevertheless a large family of imperative loops cannot be processed in parallel using only standard requests or GSRs the whole family of DO-loops where independency between successive iterations cannot guaranteed at compile time. For example, loop 1 belongs to this family.

In [Je86], we pointed out that the performance of a DSPA processor on this family of loops only depends on the effective dependencies on data. For example, when very rare actual dependencies occur between not very distent iterations during the execution of loop 1, performance of a DSPA processor is only limited by the theoretical memory throughput. It is desirable to also execute these loops in parallel on an ASM. Notice that this potential parallelism cannot be managed at compile time; it must be managed at execution time.

1.3.2 DO_ACROSS construction

Cytron [Cy86] proposed the DO_ACROSS mechanism for processing loops of the previous family in parallel on a TCM. Its proposition consists in inserting delays between the beginning of the successive iterations to ensure that the dependencies are respected; the execution of an iteration is deferred until some event has occured in a previous iteration. This technic can be applied with success to many loops.

```
      DO 100 I=1,N
      delay (d*(I-1))
      S1
      S2
      ..
      Ss
100 CONTINUE
```

E.g. let us consider an imperative loop which body can be divided into two successive sequences R and S . Let us suppose that there are no dependencies between the iteration i and the sequence R(i+1), but that there may be dependencies between S(i) and S(i+1). The execution of the iteration i+1 may begin independently from the completing of the iteration i. S(i+1) cannot be executed before the completing of S(i). The maximum speed-up which can be reached may be modelized by $SP=T_R/T_S$ where T_R (resp. T_S) is the execution time of the sequence R (resp. S). If the execution time of the sequence S is short besides the execution time of the sequence R, then a very high degree of parallelism can be obtained during the execution of the loop. When the number of processors is not greater than SP, all the processors can be saturated during the execution of the loop.

Unfortunately, in many cases this technic is not sufficient —when T_R is not large besides T_S (as in loop 1). Moreover, as no specific hardware has been proposed for the implementation of the DO_ACROSS construction, synchronization sections may be quite time consuming.

1.3.3 Slave requests and master requests

We now introduce the concepts of slave requests and master requests.

Definition :

□ A X-slave request from processor I cannot be P-issued until processor I has got a mark from processor I-X mod K —K is the number of processors.

□ When a X-master request is P-issued by processor I, a mark is sent to processor I+X mod K.

Notice that a X-slave request cannot be P-issued if there is no corresponding X-master request. On the other hand, a X-slave request must be associated to every X-master request. Processor I cannot P-issued a X-slave request before processor I-1 mod K has P-issued the corresponding X-master request.

The use of X-slave requests and X-master requests implies a static distribution of the loop iterations. It is straightforward that the use of X-slave requests and X-master requests covers approximately the same application domain of the DO_ACROSS construct.

Let us notice that the use of 1-slave requests and 1-master requests covers the most useful application domain of the DO_ACROSS constructs :

□ dependencies between successive iterations of the same loop

□ pipelined tasks on the distincts processors — data are successively updated by successive tasks for example.

Only 1-slave requests and 1-master requests will be considered for implementation in section III.2. From now, they are referred to respectively as slave requests and as master requests.

1.3.4 An example of use of master requests and slave requests

Let us consider the previously cited loop 1.

```
    DO 1 I=1,N
  1 A(P(I)) = A(Q(I))
```

The iterations are spread among the processors :

iterations 1, K+1, 2K+1, .. to processor 0

iterations 2, K+2, 2K+2, .. to processor 1

..

iterations K, 2K, 3K, .. to processor K-1

This loop may be divided in two parts :

— The vectors P and Q are read in the shared memory and stored in a local memory of the processors. (This part can be coded with standard requests).

— The indirect accesses through vectors Q and P are then performed :

READ of A(Q(I)) ($1<I\leq N$) is a slave request

WRITE of A(P(I)) ($1\leq I<N$) is a master request.

READ of A(Q(1)) and WRITE of A(P(N)) are standard requests.

The sequential signification of the loop is guaranteed :

for 1<I≤N, READ of A(Q(I)) cannot be P-issued before WRITE of A(P(I-1)) has been P-issued.

We shall see that when using the hardware mechanism proposed in the next section, a more efficient distribution of the iterations of the loop 1 among the processors is given by :

iterations 1 to L on processor 0,
iterations L+1 to 2L on processor 1
..
iterations (K-1)*L+1 to K*L on processor K-1
iterations K*L+1 to (K+1)L on processor 0
..

In the second part of the loop, the sequence of memory requests executed by processor I is has followed (Y=(a*K+I-1)*L):

Dummy slave request
READ A(Q(Y+1)) (standard)
WRITE A(P(Y+1)) "
READ A(Q(Y+2)) "
WRITE A(P(Y+2)) "
..
READ A(Q(Y+L)) "
WRITE A(P(Y+L)) "
Dummy master request

2 An efficient P-Issuing hardware mechanism

Very simple synchronization concepts have been proposed in the previous section. In this section, we propose very efficient hardware mechanisms to implement these synchronization primitives.

2.1 About the need for a multiple P-issuing on a single processor

Let us consider loop 1, if iteration I and iteration I+1 are executed by two distinct processors, the WRITE of A(P(I)) must always be P-issued **strictly** before the READ of A(Q(I+1)). If a single memory request can be P-issued by a processor in one cycle then a processor can at most execute an iteration every 2K cycles. So at most one iteration every two cycles can be executed by the machine —even with an infinity of processors.

To enable better performance, multiple P-issuing by a single processor is necessary.

2.2 P-issuing in parallel on a single processor

The respect of the condition (A) in theorem 1 enforces the avoiding of deadlocks on the memory. Here we propose a hardware mechanism which forces the respect of this condition. An issuing unit (IU) is added to the address computation unit in each processor (fig.5).

At cycle t, the address unit of processor j deposits (or not) a request in the request G-network and sends the label of the requested memory bank B(j,t) and some added information to IU(j) (for a clearer presentation, we assume here that this information is not coded) :

- Effective sending of a request ? (yes/no) : 1 bit R(j,t)
- Is it a GSR ? (yes/no) : 1 bit G(j,t)
- Is it a slave request ? (yes/no) : 1 bit S(j,t)
- Is it a master request ? (yes/no) : 1 bit M(j,t)

IU (j) in processor j is built with four cascaded elements :

A) A decoder

B) An input buffer INBUF(j) (K+3 bit width)

C) A FIFO queue FIU(j) (K+3 bit width)

D) An output buffer OUTBUF(j) (K+3 bit width) and a counter C(j)

A) At cycle t, the decoder converts the number of the requested memory bank B(j,t) in a vector $b_{in}(.,j,t+1)$ of K bits where $b_{in}(i,j,t+1)=0$ if $i \neq B(j,t)$ and $b_{in}(B(j,t),j,t+1)= R(j,t)$.
Bits G(j,t), S(j,t), and M(j,t) flow through this stage of the IU without being changed : $G_{in}(j,t+1)=G(j,t)$, $S_{in}(j,t+1)=S(j,t)$ and $M_{in}(j,t+1)=M(j,t)$

B) At cycle t, the input buffer receives the vector $b_{in}(.,j,t)$ and the bits $G_{in}(j,t)$,$S_{in}(j,t)$ and $M_{in}(j,t)$ from the decoder (INPUT(j,t)). $b_{cont}(.,j,t)$, $G_{cont}(j,t)$, $S_{cont}(j,t)$ and $M_{cont}(j,t)$ represents the content of the input buffer (CONT(j,t)).
CONT(j,t) is written in the FIFO queue FIU(j) at the condition W(j,t)=1 where

$$W=\text{nonnull}(\text{Empty}(j,t) + b_{in}(0,j,t)).b_{cont}(0,j,t) + \ldots b_{in}(K-1,j,t).b_{cont}(K-1,j,t) + S_{in}(j,t).S_{cont}(j,t) + M_{in}(j,t).M_{cont}(j,t) + G_{in}(j,t) + G_{cont}(j,t))$$

+ is the binary or

nonnull =1 if one bit in the input buffer is non-zero

Empty(j,t)=1 when the FIFO FIU(j)is empty

At the end of the cycle, the new content of the input buffer is given by

CONT(j,t+1) <-- not(W(j,t)).CONT(j,t) + INPUT(j,t)

—The input buffer is used to built vectors of bits b. Each non-null bit in b is associated with a memory request. Memory requests associated with a vector b are destinated to distinct memory banks.
—At most, one master request and/or one slave request are represented in a vector b.
—GSRs must be P-issued alone because of the very thin synchronization of the processors on the GSRs.
—Notice that a request to memory has to be P-issued as soon as possible. When the FIFO FIU is empty, the input buffer is written in the FIU.

C) FIU is a classical FIFO queue. This FIFO queue is assumed to accept a READ and a WRITE on every cycle.

D) At cycle t, OUTBUF(j) contains a vector of bits $b_{out}(.,j,t)$ and three bits $G_{out}(j,t)$, $S_{out}(j,t)$ and $M_{out}(j,t)$.
A bit V(j,t) indicates if OUTBUF(j) is valid at cycle t.
Counter(j,t) is the value of C(j) at cycle t.

/*Counter(j,t) represents the difference between the number of master requests that have been P-issued by processor j-1 mod K and the number of slave requests that have been P-issued by processor j*/

Inc(j,t) is a bit sent by IU(j-1 mod K) on cycle t-1

/*Inc(j,t)= 1 iff a master request has been P-issued on cycle t-1 by processor j-1 mod K*/

A new value of OUTBUF(j) is read on FIU(j) on condition not(V(j,t)) + Q(j,t)
where $Q(j,t)=(G_{out}(j,t)=0).(S_{out}(j,t)=0) + (G_{out}(j,t)=1$, for all the processors$) + S_{out}(j,t).(Counter(j,t)>0)$

/*when Q(j,t)=1 and V(j,t)=1, the requests represented in $b_{out}(.,j,t)$ may be P-issued*/

The bit $b_{out}(i,j,t).V(j,t).Q(j,t)$ is sent to MSU(i) —see section II.2.
New values of V(j,t+1), Counter(j,t+1) and Inc(j,t+1) are given by :

$Counter(j,t+1)= Counter(j,t) + Inc(j,t) - V(j,t).Q(j,t).S_{out}(j,t)$ (+ is the integer addition)

$Inc(j+1 \bmod K, t+1) = V(j,t).Q(j,t).M_{out}(j,t)$

V(j,t+1) = V(j,t).Q(j,t).not(FIU empty at cycle t) + not(Q(j,t)).V(j,t)

General comments

□ One can easily verify that the P-issuing respects condition (A) in theorem 1 and also respects the definition of standard requests, slave requests, master requests and GSRs.

□ Several memory requests may be P-issued at the same time by a single processor, these requests are destinated to distinct memory banks.

□ The four stages of the IU are very simple devices. It clearly appears that the delay to cross stages A) and B) is very limited. We may assume that this delay will be shorter than the FIFO queue clock. The

delay to cross the stage D) is approximately equal to the FIFO queue clock : decision to read the next word in the FIFO queue depends on the previous read word, but this decision is available after the crossing of a very few gates.
We hope that a specific device for the IU will be implemented in current Cmos technology with a clock around 50 ns.

- The whole control of request network (i.e the MSUs of the logical memory banks and the IUs of the processors) may be centralized on a single board. For a 16*16 GREEDY network, this control board will have to receive around 8 bits of information from each processor and to send 5 bits to each output of the request GREEDY network.

The time lost for synchronizing the processors in the ASM is very short :

- the address computation goes on until the address computation unit is waiting for some data.
- due to its parallel P-issuing potential, the IU is not a bottleneck.
- If DSPA processors are used, the sequencing of the instructions goes on until the sequencer is waiting for some data (or the instruction FIFO queue of some FU is full).

We believe that the proposed mechanism is sufficient to efficiently execute in parallel the major part of the algorithms used in scientific computing. Nevertheless other synchronization mechanisms may be implemented by software using the ReadModifyWrite instruction.

3 When loop 1 becomes a parallel loop

We have simulated by software an ASM designed around DSPA processors where the INs are G-networks, and the processors are synchronized through the mechanism proposed in the previous section. Our goal is not to present extensive results but only to point out that some important parallelism may be managed at execution time.

The following characteristics have been assumed :

- Same number of processors and logical memory banks
- 8 physical banks by logical bank
- A physical memory bank is busy during T_{busy}=6 cycles by a request
- Maximum depth of the FIFO queues of requests and data : 16
- A hardware mechanism for RAW hazards detection is associated to every physical memory bank. No more than 16 WRITEs waiting for data in this mechanism
- Maximum depth of the FIFO queues in the G-networks : 24
- Delay for writing a FIFO queue : 1 cycle
- Delay for reading a FIFO queue : 1 cycle

□ Delay for the computation of an integer addition by the address unit : 1 cycle

We have simulated the loop 1 coded as suggested in the second part of section III.1.4 with L=16.

Table 1 represents the average number of cycles needed for executing one iteration of loop 1 when P[I] and Q[I] are randomly selected in [1,M].

nb proc	1	2	4	8	16
M=100	4.40	2.70	2.05	1.81	1.61
M=1,000	4.25	2.13	1.17	0.81	0.65
M=30,000	4.25	2.13	1.09	0.71	0.51

Table 1

Table 1 shows the efficiency of our P-issuing mechanism. The performance of the ASM increases when the probability of dependencies between not very distent iterations decreases. If parallel P-issuing had not been possible then the average time to execute one iteration of loop 1 would always have exceeded 2 cycles, it is only equal to 0,51 cycles with 16 processors for M=30,000! These 0,51 cycles have to be compare with the minimum delay to complete the sequence (READ of A(Q(I)), WRITE of A(P(I))) which is 23 cycles in our simulator —the average delay may be two or three times longer. The execution of more than 100 iterations of loop 1 are concurrently executed in the machine!

Conclusion

Need for performance on both vector and non-vector algorithms exists in many scientific applications. In many cases, the performance achieved by a single processor is not sufficient. The GREEDY network is introduced to allow a realistic implementation of an Address Synchronized Multiprocessor —a global hardware detection of RAW hazards is performed. The control of this network is very original for use in TCM : a producer can always send its data into a FIFO queue, but a consumer must explicitly read the data onto FIFO queues. When using GREEDY networks to access the memory from the processors, the actual throughput of an interleaved and pipelined memory is very close to its theoretical throughput.

Synchronization of the processors may be done through the consuming of the memory requests by the logical memory banks. Four types of memory requests have been introduced. Standard requests are used to perform the accesses on which no synchronization is needed. Global Synchronization Requests

are used to implement global join. The introduction of the master requests and slave requests allows to process in parallel the DOACROSS loops [Cy86]. These four types of memory requests allow to implement the synchronization sections which are the most frequently needed in scientific computing.

We have proposed a very efficient and cost-effective hardware mechanism to implement the synchronization of the processors through these four types of requests, the P-issuing mechanism. Time lost during the synchronization of the processors via memory requests is very short because the sequencing of the instructions in the processors may continue —even address computation goes on.

Efficiency of our proposition has been proven by simulations : loop 1 is a sequential loop for all the existing supercomputers, but at execution a very high degree of parallelism may be managed on this loop by our mechanisms.

The hardware needed for implementing the GREEDY network and the P-issuing unit with a basic clock around 50 ns are relevant from current Cmos technology. Nevertheless simulations have proven that an ASM with a basic clock in the 50 ns range would achieve better performance on many non-vector codes (e.g. loop 1 and 2) than the current state-of-the-art supercomputers.

Bibliography

[Bu71] P.Budnick, D.J.Kuck "The organization and use of parallel memories" IEEE Transactions on Computers, Vol.C-20, pp1566-1569, Dec.1971

[Co86] K.Courtel, DESS microelectronique report June 1986 University of Rennes.

[Cy86] R.G.Cytron "Doacross: beyond vectorization for multiprocessors (Extended abstract)" International Conference on Parallel Processing 1986, pp836-844

[Ga83] D.Gajski, D.Kuck, D.Lawrie, A.Sameh, "Cedar : a large scale multiprocessor", International Conference on Parallel Processing 1983, pp521-529

[Go83] A.Gottlieb & al., "The NYU Ultracomputer - Designing an MIMD shared memory parallel computer" IEEE Transactions on Computers, Vol. C-32, pp175-189, feb.1983

[Hw84] K.Hwang, F.A.Briggs, *Computer architecture and parallel processing*, Mac Graw Hill 1984

[Je86] Y.Jégou, A.Seznec, "Data Synchronized Pipeline Architecture : Pipelining in Multiprocessor Environment" Proceedings of the 1986 International Conference on Parallel Processing pp487-494; also Journal of parallel and distributed computing, pp508-526 dec.1986

[La75] D.H.Lawrie,"Access and alignment of data in an array computer", IEEE Transactions on Computers, vol C-24, pp.1145-1155, dec.1975.

[Pf85] Pfister & al "The IBM Research Parallel Processor Prototype (RP3): introduction and architecture" International Conference on Parallel Processing 1985.

[Se86] A.Seznec, Y.Jégou "Address Synchronized Multiprocessor Architecture" rapport INRIA 527 Juillet 1987.

[Se87a] A.Seznec, Y.Jégou "Optimizing memory throughput in a tightly coupled multiprocessor" Proceedings of the 1987 International Conference on Parallel Processing, pp344-346

[Se87b] A.Seznec, Y.Jégou, J.Lenfant "A new storage scheme for optimizing both vector and non-regular accesses to memory in a tightly coupled multiprocessor" submitted.

[Se87c] A.Seznec, "Contribution à l'étude des multiprocesseurs fortement pipelinés" thèse d'état, Juin 1987, Université de Rennes I

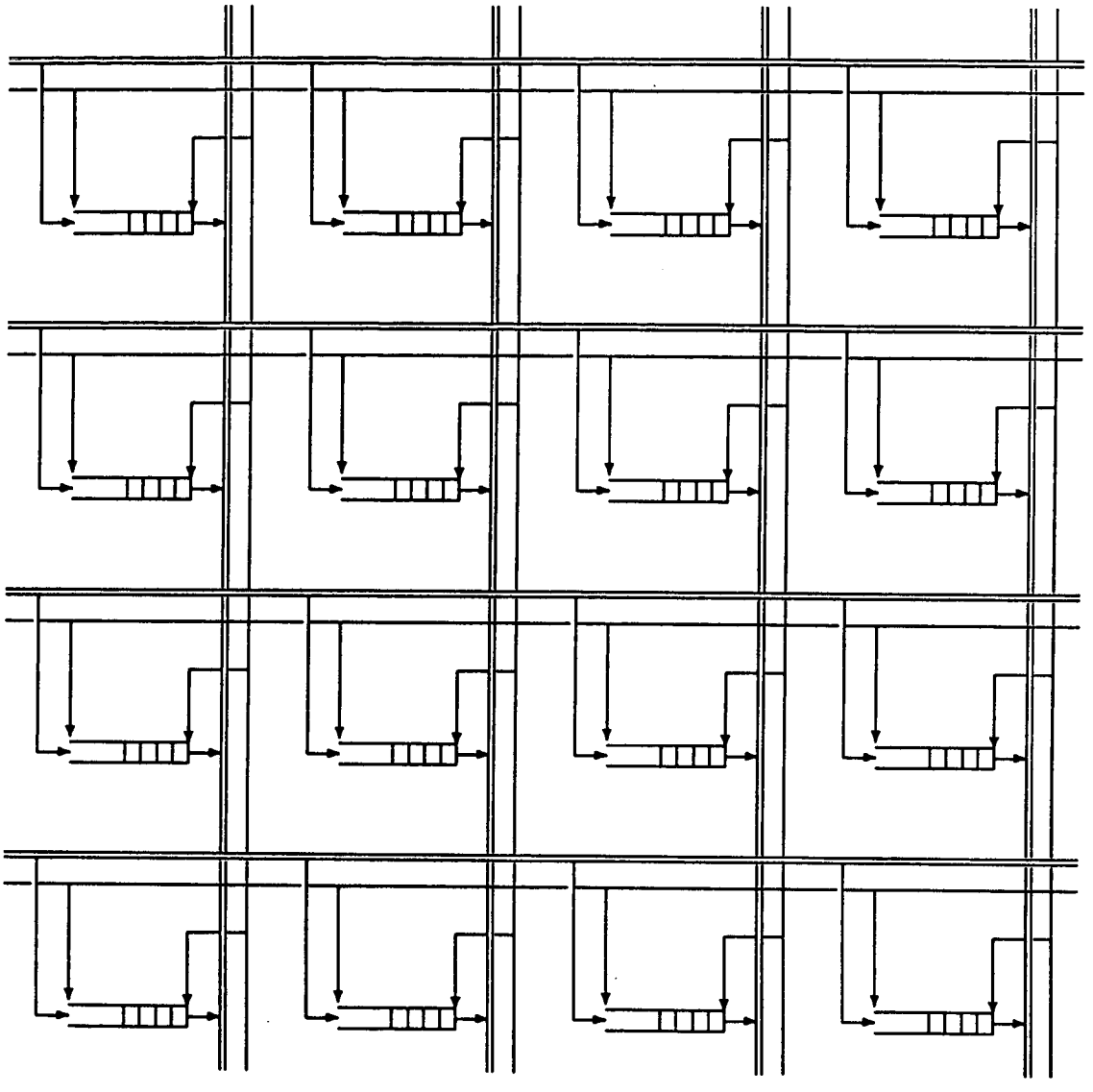
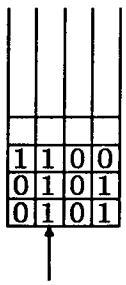
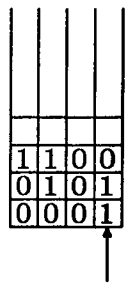


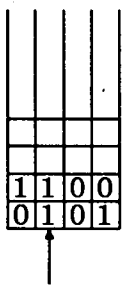
figure 1 : a 4*4 GREEDY network



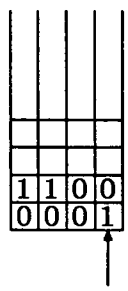
t=0; request from P1



t=1; request from P3



t=2; request from P1



t=3; request from P3

figure 2: load of the requests by the MSU

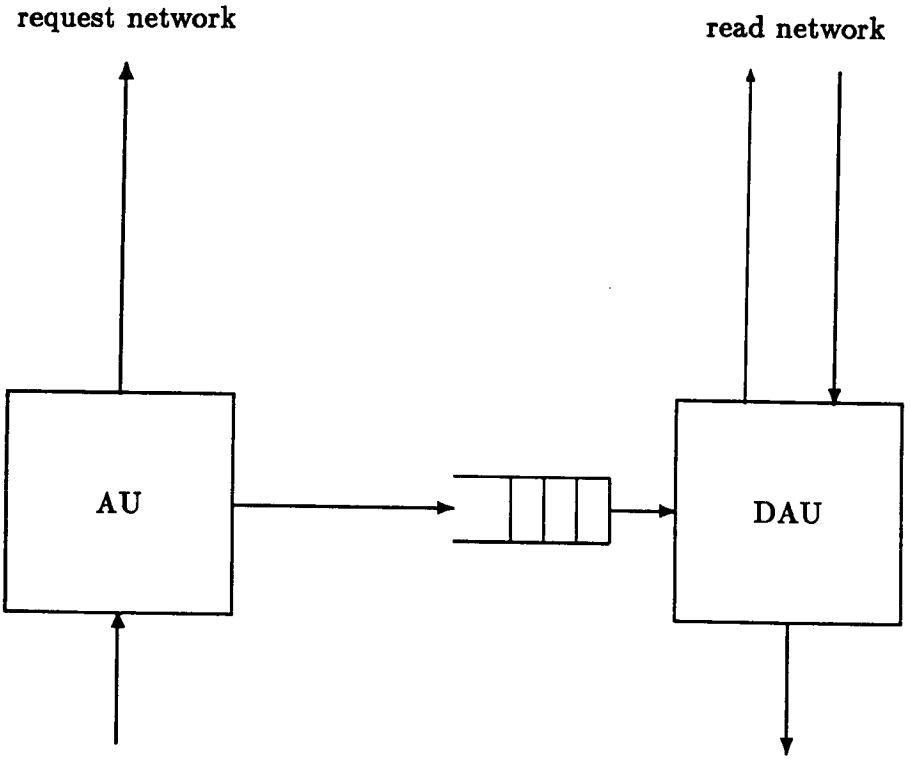


figure 3 : Data Acquisition Unit

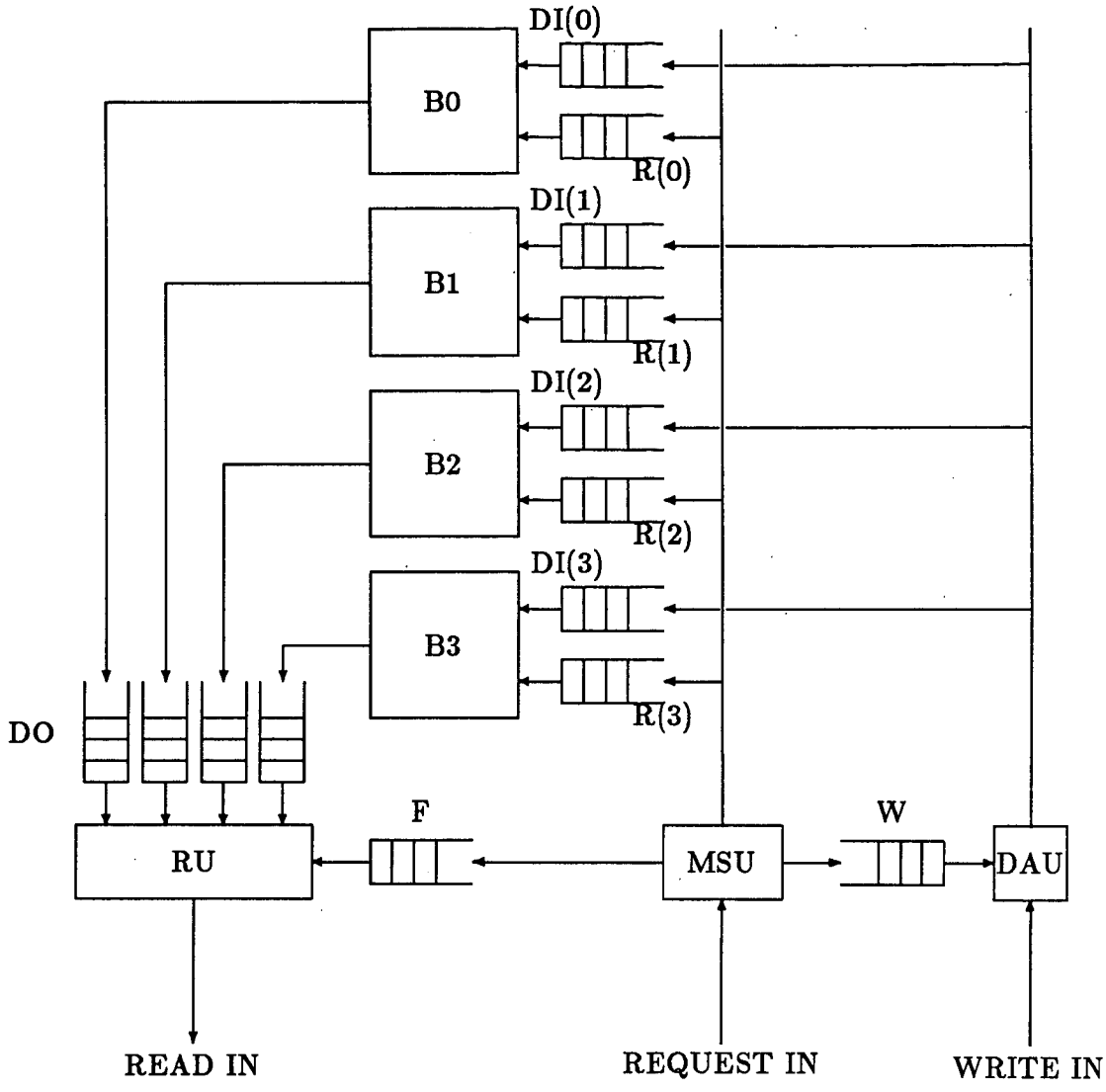


fig.4 : an optimized structure of a logical memory bank

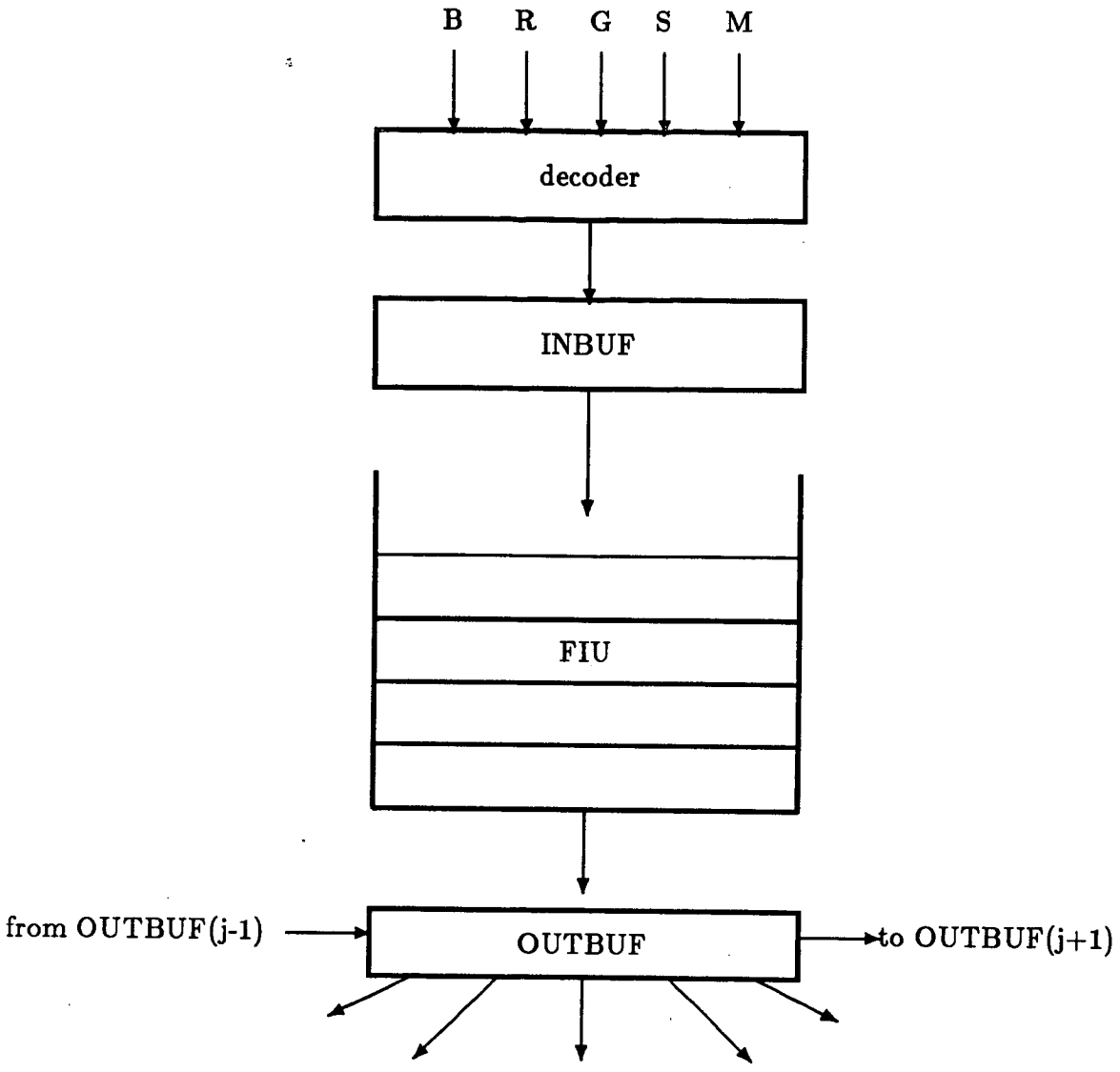


figure 5: the issuing unit

Imprimé en France
par
l'Institut National de Recherche en Informatique et en Automatique